This report presents the results of three different graph search algorithms – Breadth First Search, Djikstra's Algorithm, and Bellman-Ford Algorithm.

Graph Traversal Algorithms

Ayushya Amitabh
CSC 22000 – ALGORITHMS
PROF. CHI HIM "TIMMY" LIU

Table of Contents

THEORY

Graph traversal (also known as graph search) refers to the process of visiting (checking and/or updating) each vertex in a graph. Such traversals are classified by the order in which the vertices are visited. The goal of a graph traversal, generally, is to find all nodes reachable from a given set of root nodes. In an undirected graph, we follow all edges; in a directed graph, we follow only out-edges.

IN THIS REPORT

In this report, I compare the results for the undirected and directed paths for each of the following algorithms:

1. Breadth First Search
2. Djikstra's Algorithm
3. Bellman Ford Algorithm

Breadth First Search

Breadth First searches are performed by exploring all nodes at a given depth before proceeding to the next level. This means that all immediate children of nodes are explored before any of the children's children are considered. BFS uses a queue structure to hold all generate but still unexplored nodes. The order in which nodes are placed on the queue for removal and exploration determines the type of search.

Pseudocode:

```
//Set all nodes to "not visited";
   q = new Queue();
   q.enqueue(initial node);
   while ( q ≠ empty ) do    {
      x = q.dequeue();
      if ( x has not been visited )  {
         visited[x] = true;
      // Visit node x !
         for ( every edge (x, y))
            if ( y has not been visited )
                  q.enqueue(y);
      // Use the edge (x,y) !!!
      }
```

Djikstra's Algorithm

Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a SPT (shortest path tree) with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has minimum distance from source.

Pseudocode:

```
//insert all nodes to the priority queue, node from has a distance 0, all
others infinity
Q = InsertAllNodesToTheQueue(d, from)
CLOSED = {} //closed nodes - empty set
predecessors = new array[d.nodeCount] //array of ancestors

while !Q.isEmpty() do
node = Q.extractMin()
CLOSED.add(node) //close the node

//contract distances
for a in Adj(node) do //for all descendants
if !CLOSED.contains(a) //if the descendatn was not closed yet
//and his distance has decreased
if Q[node].distance + d[node][a] < Q[a].distance
//zmen prioritu (vzdalenost) uzlu
Q[a].distance = Q[node].distance + d[node][a]
//change its ancestor
predecessors[a] = node

return predecessors
```

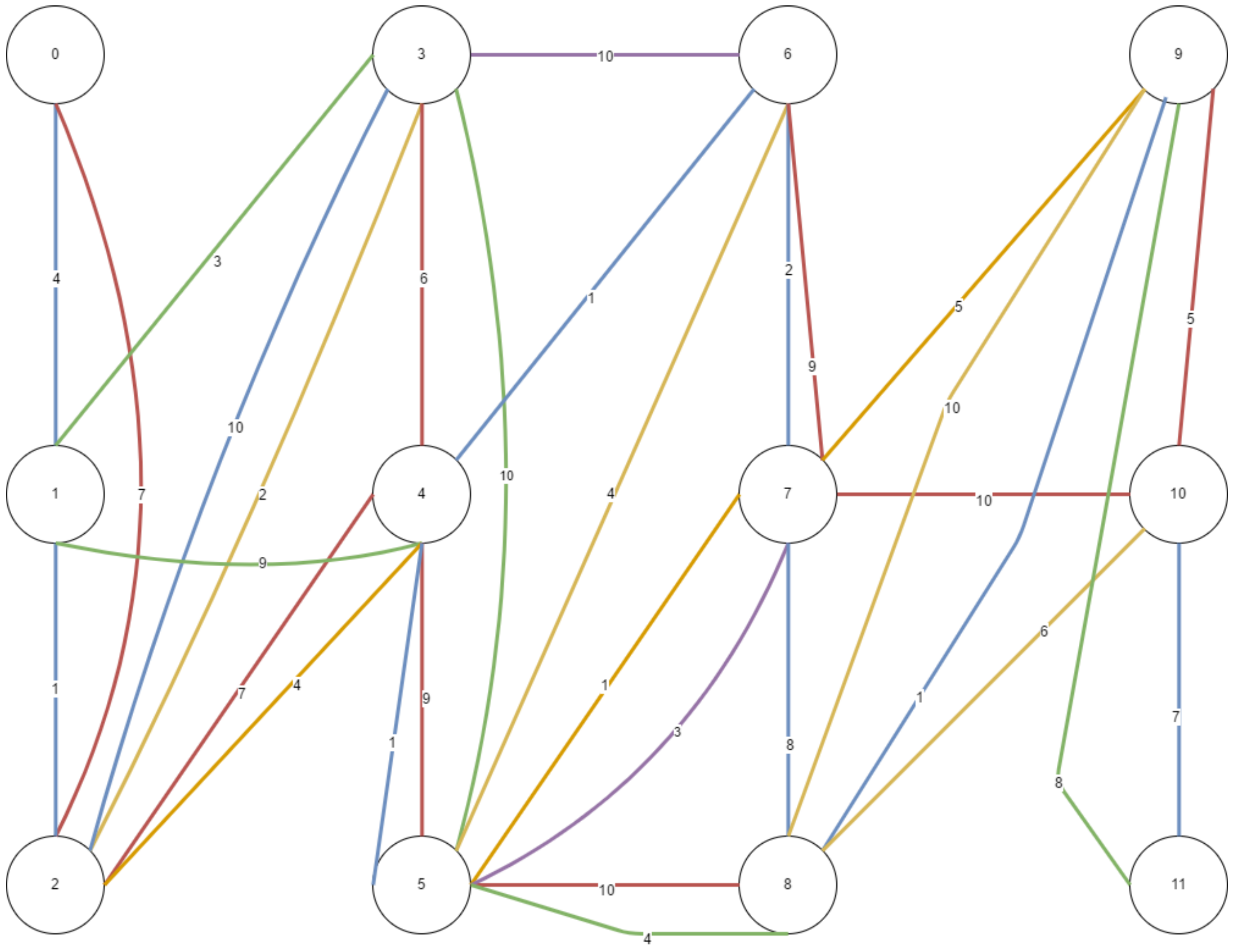
Bellman Ford Algorithm

Bellman-Ford algorithm is a procedure used to find all shortest path in a graph from one source to all other nodes. The algorithm requires that the graph does not contain any cycles of negative length, but if it does, the algorithm is able to detect it. The Bellman-Ford algorithm is based on the relaxation operation. The relaxation procedure takes two nodes as arguments and an edge connecting these nodes.

Pseudocode:

```
for i in 1 to |U| do
distance[i] = +inf
predecessors[i] = null

distance[source] = 0

for i in 1 to (|U| - 1) do
for each Edge e in Edges(G) do
if distance[e.from] + length(e) < distance[e.to] do
distance[e.to] = distance[e.from] + length(e)
predecessors[e.to] = e.from

for each Edge e in Edges(G) do
if distance[e.from] + length(e) < distance[e.to] do
error("Graph contains cycles of negative length")

return predecessors
```

THE INPUT: VISUAL



BREAKING DOWN THE CODE

Functionality for all the algorithms are based off of the first two classes listed below – Graph and Queue. I worked on Breadth First Search first and so Djikstra's implementation and the Bellman Ford implementation were restructured with little changes to the logic and structuring.

Class: Graph

```
class Graph {
private:
	int n; /// n is the number of vertices in the graph
	int **A; /// A stores the edges between two vertices
public:
	Graph(int size = 2);
	~Graph();
	bool isConnected(int, int);
	void addEdge(int u, int v);
	void BFS(int);
};

Graph::Graph(int size) {
	int i, j;
	if (size < 2) n = 2;
	else n = size;
	A = new int*[n];
	for (i = 0; i < n; ++i)
		A[i] = new int[n];
	for (i = 0; i < n; ++i)
		for (j = 0; j < n; ++j)
			A[i][j] = 0;
}

Graph::~Graph() {
	for (int i = 0; i < n; ++i)
		delete[] A[i];
	delete[] A;
}

/*****************************************************
Checks if two given vertices are connected by an edge
@param u vertex
@param v vertex
@return true if connected false if not connected
*****************************************************/
bool Graph::isConnected(int u, int v) {
	return (A[u][v] == 1);
}

/****************************************************
adds an edge E to the graph G.
@param u vertex
@param v vertex
****************************************************/
void Graph::addEdge(int u, int v) {
	A[u][v] = A[v][u] = 1;
}

/****************************************************
performs Breadth First Search
@param s initial vertex
****************************************************/
void Graph::BFS(int s) {
	Queue Q;

	/** Keeps track of explored vertices */
	bool *explored = new bool[n + 1];

	/** Initailized all vertices as unexplored */
	for (int i = 1; i <= n; ++i)
		explored[i] = false;

	/** Push initial vertex to the queue */
	Q.enqueue(s);
	explored[s] = true; /** mark it as explored */
	cout << "Breadth first Search starting from vertex ";
	cout << s << " : " << endl;

	/** Unless the queue is empty */
	while (!Q.isEmpty()) {
		/** Pop the vertex from the queue */
```

```
			int v = Q.dequeue();

			/** display the explored vertices */
			cout << "Found node : " << v << endl;

			/** From the explored vertex v try to explore all the
			connected vertices */
			for (int w = 1; w <= n; ++w)

				/** Explores the vertex w if it is connected to v
				and and if it is unexplored */
				if (isConnected(v, w) && !explored[w]) {
					cout << v << '\t' << w << "<<== Connected and Not Explored" ;
					/** adds the vertex w to the queue */
					Q.enqueue(w);
					cout << "Adding " << w << " to queue" << endl;
					/** marks the vertex w as visited */
					explored[w] = true;
				}
		}
		cout << endl;
		delete[] explored;
}
```

Class: Queue

```
class Queue {
private:
	node *front;
	node *rear;
public:
	Queue();
	~Queue();
	bool isEmpty();
	void enqueue(int);
	int dequeue();
	void display();

};

void Queue::display() {
	node *p = new node;
	p = front;
	if (front == NULL) {
		cout << "\nNothing to Display\n";
	}
	else {
		while (p != NULL) {
			cout << endl << p->info;
			p = p->next;
		}
	}
}

Queue::Queue() {
	front = NULL;
	rear = NULL;
}

Queue::~Queue() {
	delete front;
}

void Queue::enqueue(int data) {
	node *temp = new node();
	temp->info = data;
	temp->next = NULL;
	if (front == NULL) {
		front = temp;
	}
	else {
		rear->next = temp;
	}
	rear = temp;
}

int Queue::dequeue() {
	node *temp = new node();
	int value;
	if (front == NULL) {
		cout << "\nQueue is Emtpty\n";
	}
	else {
		temp = front;
		value = temp->info;
		front = front->next;
		delete temp;
	}
	return value;
}

bool Queue::isEmpty() {
	return (front == NULL);
}
```

Struct: Node

```
struct node {
	int info;
	node *next;
};
```

Class: Djikstra

```
// Functions are same as for Graph but modified and broken down for simplicity and to
match the Djikstra Algorithm

class Dijkstra {
private:
	int adjMatrix[15][15];
	int predecessor[15], distance[15];
	bool mark[15]; //keep track of visited node
	int source;
	int numOfVertices;
public:
	void read(); //Gets Data
	void initialize(); // Initializes the relaxation method
	int getClosestUnmarkedNode();
	void calculateDistance();
	void output();
	void printPath(int);
};
```

Class: Bell_Ford

```
// Functions are same as for Graph but modified and broken down for simplicity and to
match the Bellman Ford Algorithm

class bell_ford
{
private:
	int n;
	int graph[MAX][MAX];
	int start;
	int distance[MAX];
	int predecessor[MAX];
public:
	void read_graph();
	void initialize();
	void update();
	void check();
	void algorithm();
};
```

RESULTS & PROCESS

Breadth First Search: Undirected

```
Breadth first Search starting from vertex 0 :
Found node : 0
0	1<<== Connected and Not Explored
Adding 1 to queue
0	2<<== Connected and Not Explored
Adding 2 to queue
Found node : 1
1	3<<== Connected and Not Explored
Adding 3 to queue
1	4<<== Connected and Not Explored
Adding 4 to queue
Found node : 2
Found node : 3
3	5<<== Connected and Not Explored
Adding 5 to queue
3	6<<== Connected and Not Explored
Adding 6 to queue
Found node : 4
Found node : 5
```

```
5	7<<== Connected and Not Explored
Adding 7 to queue
5	8<<== Connected and Not Explored
Adding 8 to queue
Found node : 6
Found node : 7
7	9<<== Connected and Not Explored
Adding 9 to queue
7	10<<== Connected and Not Explored
Adding 10 to queue
Found node : 8
Found node : 9
9	11<<== Connected and Not Explored
Adding 11 to queue
Found node : 10
Found node : 11
```

Breadth First Search: Directed

```
Breadth first Search starting from vertex 0 :
Found node : 0
0	1<<== Connected and Not Explored
Adding 1 to queue
0	2<<== Connected and Not Explored
Adding 2 to queue
Found node : 1
Found node : 2
2	1<<== Connected but Already Explored
2	3<<== Connected and Not Explored
Adding 3 to queue
2	4<<== Connected and Not Explored
Adding 4 to queue
Found node : 3
3	1<<== Connected but Already Explored
3	2<<== Connected but Already Explored
3	4<<== Connected but Already Explored
3	6<<== Connected and Not Explored
Adding 6 to queue
Found node : 4
4	1<<== Connected but Already Explored
4	2<<== Connected but Already Explored
4	5<<== Connected and Not Explored
Adding 5 to queue
Found node : 6
6	4<<== Connected but Already Explored
6	5<<== Connected but Already Explored
6	7<<== Connected and Not Explored
Adding 7 to queue
```

```
Found node : 5
5	3<<== Connected but Already Explored
5	4<<== Connected but Already Explored
5	7<<== Connected but Already Explored
5	8<<== Connected and Not Explored
Adding 8 to queue
Found node : 7
7	5<<== Connected but Already Explored
7	6<<== Connected but Already Explored
7	10<<== Connected and Not Explored
Adding 10 to queue
Found node : 8
8	5<<== Connected but Already Explored
8	7<<== Connected but Already Explored
8	9<<== Connected and Not Explored
Adding 9 to queue
8	10<<== Connected but Already Explored
Found node : 10
10	9<<== Connected but Already Explored
10	11<<== Connected and Not Explored
Adding 11 to queue
Found node : 9
9	7<<== Connected but Already Explored
9	8<<== Connected but Already Explored
9	11<<== Connected but Already Explored
Found node : 11
```

Djikstra’s: Undirected

```
0..0->0
0..1..->4
0..1..2..->5
0..1..3..->7
0..1..2..4..->9
0..1..2..4..5..->10
0..1..2..4..6..->10
0..1..2..4..5..7..->13
0..1..2..4..5..8..->14
0..1..2..4..5..7..9..->18
0..1..2..4..5..8..10..->20
0..1..2..4..5..7..9..11..->26
```

Djikstra’s: Directed

```
0..0->0
0..1..->4
0..2..->7
0..2..3..->9
0..2..3..4..->12
```

```
0..2..3..4..5..->21
0..2..3..6..->19
0..2..3..6..7..->21
0..2..3..4..5..8..->31
0..2..3..4..5..8..9..->32
0..2..3..6..7..10..->31
0..2..3..6..7..10..11..->38
```

Bellman Ford: Undirected

```
Enter the no.of nodes in the graph:: 12
Enter the start vertex::0

There is no negative weight cycle
****** The final paths and the distacnes are ******
path for node 0 is ::
0
distance is 0
path for node 1 is ::
1
distance is 4
path for node 2 is ::
2
distance is 7
path for node 3 is ::
2->3
distance is 9
path for node 4 is ::
2->3->4
distance is 12
path for node 5 is ::
2->3->4->5
distance is 21
path for node 6 is ::
2->3->6
distance is 19
path for node 7 is ::
2->3->6->7
distance is 21
path for node 8 is ::
2->3->4->5->8
distance is 31
path for node 9 is ::
2->3->4->5->8->9
distance is 32
path for node 10 is ::
2->3->6->7->10
distance is 31
```

```
path for node 11 is ::
2->3->6->7->10->11
distance is 38
```

Bellman Ford: Directed

```
Enter the no.of nodes in the graph:: 12
Enter the start vertex::0

There is no negative weight cycle and
****** The final paths and the distacnes are ******

path for node 0 is ::
0
distance is 0
path for node 1 is ::
1
distance is 4
path for node 2 is ::
1->2
distance is 5
path for node 3 is ::
1->3
distance is 7
path for node 4 is ::
1->2->4
distance is 9
path for node 5 is ::
1->2->4->5
distance is 10
path for node 6 is ::
1->2->4->6
distance is 10
path for node 7 is ::
1->2->4->5->7
distance is 11
path for node 8 is ::
1->2->4->5->8
distance is 14
path for node 9 is ::
1->2->4->5->8->9
distance is 15
path for node 10 is ::
1->2->4->5->8->10
distance is 20
path for node 11 is ::
1->2->4->5->8->9->11
distance is 23
```